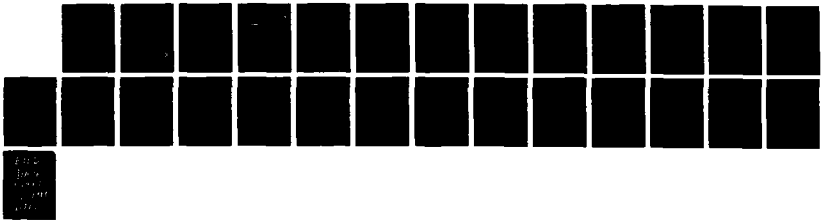
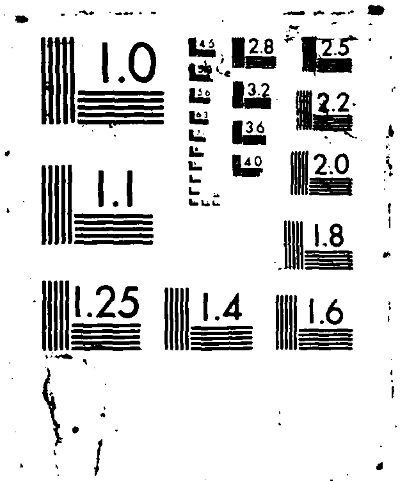
AD-A192 264 SIMS: SINGLE INTERFACE TO MULTIPLE SYSTEMS(U) 1/1
UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY
INFORMATION SCIENCES INST J PAVLIN ET AL. FEB 88
UNCLASSIFIED ISI/RR-88-200 MDA903-81-C-0335 F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(4)

AD-A192 264

DTIC FILE COPY

ISI Research Report
ISI/RR-88-200
February 1988

University of Southern California

Jasmina Pavlin
Raymond L. Bates

SIMS: Single Interface to Multiple Systems

INFORMATION SCIENCES INSTITUTE ISI
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

88 3 21 092

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-88-200	5. MONITORING ORGANIZATION REPORT NUMBER(S) ---------------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute; 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION ---------------
6c. ADDRESS *(City, State, and ZIP Code)* 4676 Admiralty Way Marina del Rey, CA 90292	7b. ADDRESS *(City, State, and ZIP Code)* ---------------
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Defense Advanced Research Projects Agency; 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903-81-C-0335
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Boulevard Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
---------------	---------------	---------------	---------------

11. TITLE *(Include Security Classification)*
SIMS: Single Interface to Multiple Systems [Unclassified]

12. PERSONAL AUTHOR(S) Pavlin, Jasmina and Bates, Raymond L.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Research Report	FROM ________ TO ________	1988, February	28

16 SUPPLEMENTARY NOTATION

17 COSATI CODES

FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
code reusability, error recovery, execution monitoring, framework, multiple software servers, hierarchical planning, software integration, user interfaces

19 ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report describes the Single Interface to Multiple Systems (SIMS) project, which is developing a tool to automatically invoke services to perform the tasks of information systems users. Most of these tasks involve multiple software servers that differ vastly in their functional capabilities, their view of the world, and their input and output languages (e.g., databases and expert systems). Before a user can determine if a server can perform the required task, he must first understand the language and services provided by each server, and then express the task in the terms used by the server. In contrast, SIMS allows the user to specify a task in a service-independent language. Our approach is dynamic in that SIMS determines which server will satisfy the request when the task is defined. SIMS performs logical integration, where the servers remain separate, but a common model makes them appear as if they were providing a single virtual service. Logical and dynamic integration allow the system of which SIMS is a part to be more easily designed and maintained.

[illegible]

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Sheila Coyazo Victor Brown	22b. TELEPHONE *(Include Area Code)* 213-822-1511; 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ISI Research Report
ISI/RR-88-200
February 1988

University of Southern California

Jasmina Pavlin
Raymond L. Bates

SIMS: Single Interface to Multiple Systems

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is sponsored by the Defense Advanced Research Projects Agency under Contract No. MDA903-81-C-0335. Any opinions, conclusions or recomendations expressed in this publication are those of the authors' and should not be interpreted as representing the official opinion of DARPA, the U.S. Government, or any person or agency connected with them.

SIMS: Single Interface to Multiple Systems

Jasmina Pavlin Raymond L. Bates

1 Introduction

Most of the tasks performed by users of information systems involve interaction with multiple software servers. A typical task may involve database retrieval, data interpretation, numerical calculation and report generation. The servers can be vastly different in their functional capabilities, view of the world and their input and output languages (e.g., databases and expert systems). The user must understand the language and services provided by each server, and express his task in the terms used by the server before he can decide whether the server provides a service required to perform the task. The application of servers is solely the user's responsibility. The system cannot help the user figure out how to perform his task, since there is no explicit representation of services provided by a server. Even when the user fully specifies his task in terms of available servers, he has to perform the task manually in all but the simplest of cases (e.g., UNIX pipes). The user's job is not only difficult, but also prone to numerous errors, due to its ad hoc nature. We term the problem facing the user the *software integration* problem. It consists of: selection and sequencing of software servers, enforcing compatibility between inputs and outputs of different servers, server invocation, execution monitoring, and result formulation.

The only approach to the software integration problem has been to build custom systems that support the integrated view of underlying servers. The advantage of custom systems is that the user can state his tasks in server-independent terms, and does not need to understand the language and the functionality of each server. Such systems are hand-crafted for the application at hand, and are built at great expense. Although they may be well polished and functional, they are very difficult to modify and generalize, because the integration is hardwired into the system. The principles underlying the integration are at best known only to the system designer, and are at worst nonexistent. Typically, when a new service is needed, the new system is built from scratch, since adding a server to the existing system or modifying an existing server to provide a desired service carries a high risk of inconsistent behavior. Thus, what should be a minor change results in a major design and programming undertaking, with little chance to utilize the experience gained from the old system.

Our goal for this work is to ensure that the versatility of accessible services bring only an advantage and not a burden to the users and designers of systems. The system should give to the user all the advantages of a custom-built system and should not suffer from the inflexibilities that hurt maintainers and future designers. The SIMS project is predicated on providing a framework for the integration of software services that result in such systems. The following features are essential for its design:

- SIMS abstracts out the differences between servers by providing a *server-independent interaction language*.
- SIMS' approach is *logical integration*, where the servers remain separate, but a common

model makes them appear as if they were providing a single virtual service.

- The integration is *principled*, that is, there is a well defined procedure for system modification when an underlying server is changed or added and this procedure is limited in scope.

- The sequencing is *dynamic*, that is, the sequence of servers that perform the task is determined when the task is defined. This is in contrast with static sequencing (determining and storing all possible mappings between tasks and servers at design time), where adding a new server may involve revision of all mappings in the system. Dynamic sequencing does not suffer from this problem, and allows for optimizations based on runtime conditions.

Software integration is central for the design of a new generation of user interfaces which will be easier to design, maintain and use, due to the high level of separation between the application services and other interface functions. In this paradigm, SIMS works as an intermediary between the interface and the application. The interface presents requests to SIMS in a canonical form (a formal language of the task domain), and SIMS provides results (or error conditions) in the same canonical form to the interface. SIMS is responsible for all interactions with application servers.

We are building a system following this paradigm in conjunction with the Integrated Interfaces project [KACZ 86] at Information Sciences Institute. The Input Analysis component of the system transforms inputs entered in various modes into requests expressed in the formal language acceptable by SIMS. Requests that involve interaction with servers are related to SIMS. The Central Coordination component gives the output of SIMS to the Presentation Planner, which decides which presentation modes to use in order to give this information to the user. Domain and system knowledge used by all components of the system is contained in the Capability Knowledge Base. The architecture of the system is shown in Figure 1.

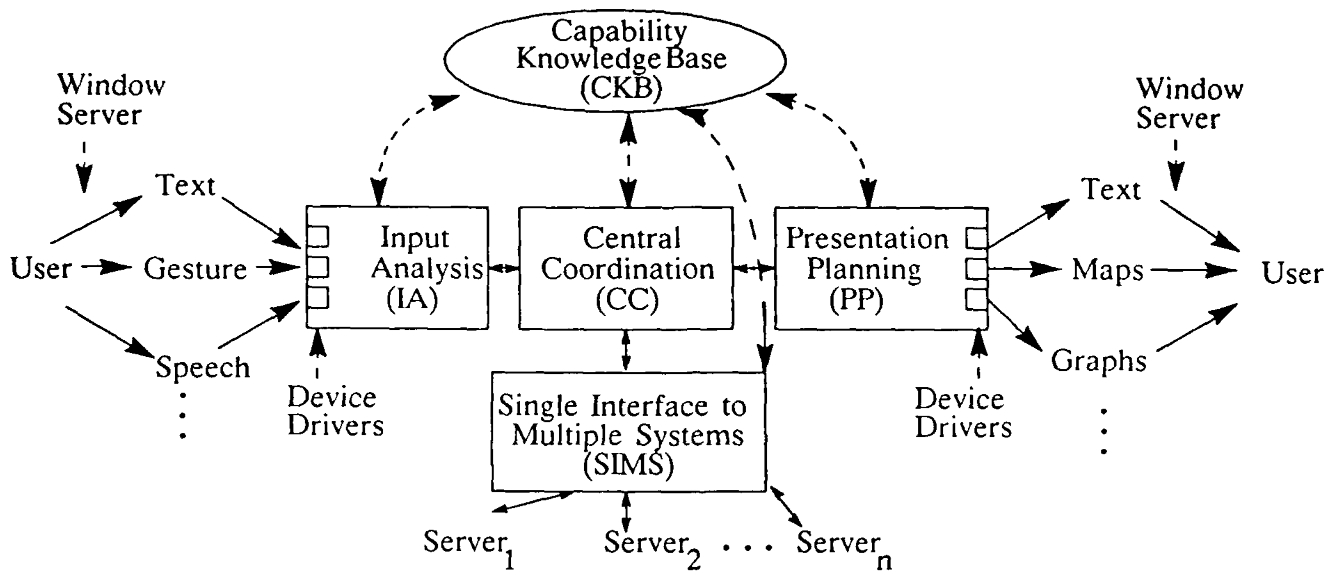


Figure 1: The Integrated Interfaces User Interface Architecture.

Design is simplified in this paradigm because of a clean separation of concerns between the interface designer and the application designers. The interface can be developed before the application, for example, as long as the specifications for the functionality exists. Also, since the functionality of application servers is accessible to SIMS, code reusability is facilitated. The main advantages of this approach, however, will come as the system evolves, since the interface can remain unchanged when underlying servers are modified. A stable and reusable interface is important not only from user's point of view, but also from software engineering standpoint, where it has been recognized that more than half of the work required in setting up the system is spent on the interface design [CARL 79]. The separation advantage works in the other direction as well. If the interface needs to be changed (e.g., customized to suit a particular class of users) the application can remain the same. Since the interface is freed from the application concerns, it can be made consistent and easier to use.

Some aspects of the general problem of software integration have been addressed by existing systems on a limited scale. The UNIX pipe utility invokes a specified sequence of single-input and single-output servers whose input and output is of a single type (byte string). The motivation for SIMS comes from the commonly heard comment of pipes' users: "This is a fine idea, if only it would go a bit further!" We plan to extend out work in several dimensions, such as:

- expressing the task in server-independent terms;
- automatically determining the combination of servers that satisfy the task;
- allowing for other control structures besides a simple sequence of servers;
- Execution monitoring and replanning around errors; and
- Reformulation of the result into the task language.

Abstract Data Types [STEM 86] provide one tool for software integration, since they represent a high-level specification of data that allows more complex structures to be passed between servers. Object-oriented languages [STEF 86] define legal communication methods between objects and thus simplify the problem of testing the compatibility of servers. Integrated database research [MOTR 80] addresses a different but related problem of combining database schemas.

Physical integration is performed, resulting in a single database. In contrast, SIMS' approach is in logical integration.

In its most general form, the task of dynamic software integration is more difficult than the general automatic program synthesis task. The program that performs the task should be synthesized and executed – the results should then be presented to the user. Fully automatic synthesis of arbitrary programs is itself too difficult, particularly because of the combinatorics involved in the search for possible transformations between the source and the target language. At least some of the following simplifications of the problem are made [FEAT 86]:

- The process is not fully automatic, i.e., it is allowed to take advice from the programmer.
- The source language is simplified at the expense of generality.
- The programming task is highly restricted.

SIMS takes an advantage of the last two simplifications, as a natural consequence of the problem it is addressing, which is software integration in the context of information services. Information services provide a highly structured but a not-trivial class of tasks for which dynamic software integration has a great practical value. The types of requests are naturally limited (e.g., to requests for objects of a certain class among which certain relations hold). For a large class of useful requests, the search space can be pruned to a small size by eliminating services that are irrelevant for the request. The number of ways in which the services can be combined for a given task is also relatively small (compared to the number of ways in which a typical program can be synthesized), since each server provides a highly specialized function. Thus, automatic software integration is both useful and tractable in the domain of information services.

Major subproblems that SIMS must address are: representation, formulation (of the request in server terms), execution monitoring, and formulating the results in terms of the request language. They are discussed in the following sections.

Section 2 describes a uniform representation for services and requests in terms of relations. A server is represented by a number of *service relations*, one for each distinct function provided by the server.

Section 3 presents our approach to the formulation problem (mapping between the request and the available servers) which consists of three steps. In the first step, service relations that are irrelevant for the request are eliminated. The second step finds a combination of service relations that satisfy the request. Multiple combinations (paths) are provided by the algorithm. The same service can be provided by multiple servers or by multiple ways to invoke the same server. The role of the last step is to find the best path to use, based on a utility model of each server. The final subsection of Section 3 describes some mechanisms for correction of requests that cannot be satisfied as formulated.

Section 4 discusses the recovery of execution errors, which is accomplished by retaining multiple paths and multiple servers within the same path. Its efficiency depends on retaining as much of the current path as possible.

Section 5 outlines our initial solution to the problem of formulating the results in the request language, once the request has been successfully satisfied.

Section 6 provides a brief summary and an indication of our immediate short-term plans for SIMS.

2 The Representation Problem

In order to mediate between the request and the available services, we need to represent the space of possible requests (the *request space*) and the space that contains a representation of the functionality of all servers (the *service space*). In addition, we need to define the mapping between the twc spaces.

In our approach, a server is represented by a collection of relations, one relation for each service provided by the server. To simplify the terminology, we will refer to service relations simply as services when no confusion could arise.

Initially, we have considered servers of two types: relational database management programs and batch application programs that provide functions without side-effects. We will relax these restrictions to programs with side-effects, interactive programs, and expert systems in the domain of information services.

A service relation of a database server corresponds to a retrievable unit of data, so there are as many service relations as there are relations in the database. In SIMS, service relations are represented by an object, and elements of this relation are attributes of the object. An application program can provide several functions, one for each entry point in the program. A function can also be viewed as a relation between its inputs and outputs. So, each entry point of an application program also defines one service relation. A database relation differs from a function in that all its attributes can be used either as inputs or as outputs, so data can be obtained even if no inputs are provided (by dumping the whole database table). The attributes of a function have distinct constraints on which of them have to be supplied as input.

The semantics of a service relation are established by defining lower-order relations between its attributes which we term *dependencies*. For example, a database relation called SHIP-DESCRIPTION with three attributes AREA, SHIP, and EMPSKD (employment schedule) can be represented by the SHIP-DESCRIPTION service and two dependencies: IN-AREA with attributes AREA and SHIP, and SHIP-EMPSKD with attributes SHIP and EMPSKD. A function that takes an area as input and returns data for generating the corresponding map can be represented by the MAP-OF-AREA service with two attributes: AREA and MAP.

Multiple servers can represent the same information differently. The simplest aspect of this is the alternative naming problem. For example, two functions can name the same data differently. In a more complicated case, a group of attributes of one relation corresponds to a single attribute of another service relation. For example, a DATE attribute of one relation can correspond to three attributes of another relation: YEAR, MONTH, and DAY. In general, semantics of data provided by different servers can be related in an arbitrary way. We provide logical integration of servers by merging the service of different servers into a single, coherent *service space*. In the service space, the differences between semantically equivalent services are abstracted out. The service space is a hierarchy of service relations. One service is a subclass of another if it has additional dependencies or additional input constraints. An important part of the merging process relies on the semantics of the real-world objects represented by data available from servers, and hence the formation of the service space cannot be fully automated, but it appears that relatively few actions (similar to those described in [MOTR 80]) have to be performed so this procedure is well defined.

User requests are represented using the same notation and the same terms (attributes) but possibly different relations from those found in the service space. The requests in the request space are also hierarchically structured. A sample request [1] might be "Select situation data in South China Sea." This request is satisfied by collecting the situation relation objects. The SITUATION relation might have AREA, MAP, SHIP, and EMPSKD attributes, with these dependencies: MAP-OF-AREA between MAP and AREA, IN-AREA between AREA and SHIP, and SHIP-EMPSKD between SHIPS and EMPSKDS.

The request space allows the representation of requests in server-independent terms. This is important for two reasons. First, the choice of service relations is often guided by implementation considerations; the resulting terms may be very different from the terms in which the user thinks about the domain. Second, a desired property of the request language is stability: changing the service space (for example, by adding a server) should involve the smallest possible change in the request language. Adding a server may involve a revision of the service space, while the request space that represents domain relations should change little or not at all when a server

[1]This is a natural language paraphrase of the request expressed in a formal language.

in the same domain is added.

3 The Formulation Problem

The formulation problem involves transforming the request into an equivalent form expressed in terms of the service relations and executable by available servers. We will assume that the request is a conjunction of predicates, equivalent to the "select/project/join" operations of relational algebra [MAIE 83].

One approach is to treat it as a min-cost path problem, and to use graph search, A^* style algorithm [NILS 80]. This approach has been taken in [KAEM 86]. A node in the graph is a set of services covering some portion of the request, and the start node is the empty set. The solution path is a sequence of services, which combine to cover all dependencies in the request and satisfy input constraints. A heuristic function with two components (called g and $\hat{h}$ in [NILS 80]) is used to generate successors of a node n. The $g(n)$ component is the cost of the path from the start (empty set of services) to node n, represented by the cardinality of the set n. The second component of the heuristic function is the estimated cost of the path from node n to the goal node, represented by the number of dependencies in the request not covered by node n. The algorithm tests each generated node for feasibility (satisfaction of input constraints). Only one path is returned by the algorithm (the first one found to be feasible). The algorithm assumes that a path will be found and makes no attempt to deal with errors.

There are several problems in this approach. One is that the heuristic function used by the algorithm is not *admissible* [NILS 80] since $\hat{h}$ is not a lower bound on h (the actual cost of the path from the current node to the goal node). A more serious problem is that the goal of the algorithm is not to find any path but to find a *feasible* path, and thus the heuristic function used is a poor indicator of progress towards the goal. Another problem is the simple notion of optimality, which assumes that all calls to all servers carry the same cost, so the cost of a path is only proportional to the length of the path. Our main objection to this approach, however, is that the feasible cover algorithm has an unrealistic view of optimality and a complete disregard for robustness. We are taking a more balanced approach to the formulation problem.

3.1 Our Approach to the Formulation Problem

Our formulation algorithm takes an advantage of the hierarchical structure of the service space. It finds relevant services to satisfy the request by first classifying a request relation in the service space, and then finding the services that subsume it [SCHM 83]. The role of the classifier is twofold: to filter out services that are irrelevant for the request (either because they do not share any dependencies with the request, or because their input constraints are not satisfied), and to provide a list of relevant services in the order of their specificity (most specific first).

Formulation proceeds in stages. At every stage, the request is reformulated to reflect what portion of it remains to be satisfied, new classification is performed and the first service returned by the classifier is added to the sequence. The service "sequence" may, in fact, contain loops, local binding, and other programming constructs. A loop is necessary to "connect" a service that provides multiple values with another service that should be selected repeatedly using these values one by one. For example, if one service provides ships, and another links a ship with an employment schedule, a loop is needed to get employment schedules for all ships. These programming constructs are needed to enrich the server languages, most of which cannot handle requests from the previous example (e.g., SQL). The result of formulation is a kind of specification, since it has a structure of a program but is not executable (until services are replaced by actual server calls).

The process continues until all request dependencies are covered. The specification defines multiple paths for two reasons. First, an optimal path cannot be determined at this stage, since the cost of a path is a function of servers providing the services, so there are no good criteria for singling out one path before the servers are chosen. Second, robustness is assured by providing an alternative path if a path fails during execution. The number of paths is typically small, so in our current implementation all paths are returned. If this is not the case, it is easy to set a desired limit on the number of paths, but multiple paths should still be retained.

3.2 Finding Servers

The goal of this last step of the formulation algorithm is to find servers that provide services found at the previous stage. The services can be thought of as subgoals that need to be achieved as a part of the request, and we use a hierarchical-style planner [WILK 84] for this task. Alternative plans correspond to different ways to obtain the same service relation. This can mean using a different server, or a different way to invoke the same server. A plan has an associated *goal* (the service it can provide), *preconditions* (conditions that must be true for the plan to begin –

for example a database being attached or a program being loaded) and *method* (exact language for providing the corresponding server call). This stage of processing focuses on optimality, so we also define the *utility* of performing a plan. The utility is a function of the characteristics of the server providing the service and of the overall system state at the time the plan is invoked. The utility of a server is characterized by its time/space requirements, its likelihood of failure, and the reliability of data it provides. The system state affects the utility of a plan as well. The utility is improved, for example, if an application program is already loaded: subsequent functions will be less expensive to invoke. On the other hand, the utility can be reduced in the cases where there is barely enough room in memory to load the program.

A general-purpose plan for satisfying a service goal is to retrieve locally available results (without invoking a server), providing that these results have been stored. A result should be stored (cached) if it is *safe* and *worthwhile* to do so [MOST 85]. A result is safe if a later invocation of the same server gives either the same or acceptably different results. Functions without side-effects are always safe, and database services are guaranteed to be safe if there are no intervening updates. Caching is worthwhile if it is cost-effective and if it is likely that the same result will be needed again. Caching is cost-effective if time/space requirements of invoking a server are greater than the time/space requirements for storing and retrieving the cached result. The likelihood that a result will be reused is based on the pattern of user's interaction with the system, and must be provided by the domain expert. Since only safe and worthwhile results are cached, the plan for using cached results always has a greater utility than the corresponding plan for calling the server.

When the planner finds candidate plans for each goal, the utility of alternative paths for satisfying the request can be determined, and the best path is then selected for execution.

3.3 Request Formulation Errors

If a request cannot be satisfied as stated, a robust approach should invoke some request repair mechanism instead of failing. There are two approaches to request formulation errors. The first approach makes a basic assumption that the user "does not know what to ask", so the system should interact with the user to help him clarify his intentions. This approach has been taken in [MOTR 87] and [TOU 82]. We take a different approach: we assume that the request is a correct (although possibly high-level) representation of the task that the user wants performed, and that a representation for requests of this class exists in the request space. So, the request cannot be satisfied only because some dependencies of the request are not contained in any services, or because the input constraints cannot be satisfied. If we represent the partial solution (combination of services that are found to satisfy some portion of the request) as a data-flow graph, then the completion of the request can be formulated as the problem of finding a connected "extension" of the request graph. From this standpoint, the problem can be approached on either syntactic or semantic grounds. Our request repair mechanism first takes a semantic approach, assuming that the error has occurred because there is an apparent attribute-type mismatch between the output of one service and input of another. Since all attributes form a type hierarchy, this hierarchy can be used to a) test whether the mismatch is real, and b) to attempt type coercion. The following cases can arise:

- The service expects a type and its subtype is supplied. In this case, no special action is necessary, since the classifier will recognize that the service is applicable for the task.
- The server expects a type and its supertype is supplied. For example, one server provides ships, and the next expects battleships. Here, a runtime test must be introduced to verify that the data provided are still appropriate, e.g., to verify that the ship is a battleship. Such test is either one of the services in the system, or its result must be supplied by the user. To determine if such a test is one of the services, the test is posed as a request to the system and the classifier is used to find the corresponding services. If the service is found, the resulting program will start with a type test, and then branch to the appropriate server call if the test is positive, or to the appropriate error condition if the test is negative.
- Two types are not in subtype-supertype relation. This problem cannot be solved in general, but a transformation of one type into the other should be attempted if the types are equivalent. The equivalence is a semantic relation that must to be marked in the service space. Two attributes are equivalent if their corresponding value pairs form a bijection in all services where they appear. For example, names and social security numbers are attributes with such a property. If an equivalent relation can be found, then the appropriate transformation (e.g., from name to the social security number) is introduced to complete the specification.

Another strategy for dealing with a request that cannot be satisfied is to satisfy "most of" the request. An approach to this problem must break down the request into identifiable subtasks and specify the portion of the request that is satisfied by each of the subtasks. This is easy to do for some requests (e.g., if the request is to give a report about ship schedules, providing ship data in some other form would do "most" of the job), but general strategies are difficult to define.

If the attempt to correct the error using semantic methods fails, a syntactic approach could be attempted. It would be reasonable to find the most compact graph that spans the request. This is a graph-theoretic problem known as the Steiner Tree Problem. However, the resulting graph may not be the intended representation of the user's request, and the extension must be verified with the user.

4 Execution Monitoring and Error Recovery

If a plan fails during execution, recovery should be attempted. For reasons of efficiency, we try to retain as much of the currently executing path as possible, by attempting the smallest change first. The smallest change is no change at all, that is, trying to execute the same plan again. This strategy is effective for some types of errors (e.g., a random error during a function invocation). If this strategy is not appropriate, the next repair mechanism is to substitute the plan by an alternative plan satisfying the same goal. An alternative plan should be rejected if it is likely to fail as well. For example, if the plan failed because the access to the remote cite containing the server was disabled by a permanent network failure, all other plans for accessing data from this cite will also fail. This is an example where a failure of one plan can have implications on other parts of the path (if there are plans involving the same cite elsewhere in the path). If such plans are to be performed in the future, they have to be replaced as well.

The situation is different if the plans have already been performed, since the planner keeps the history of partial results during the execution of one path. Saving partial results during the satisfaction of one request is worthwhile in almost all cases, since the results are definitely needed to satisfy the request. The planner must, however, test whether a result of the plan is safe to reuse instead of invoking the plan again. For example, if a database table has been updated during the execution, the plan for accessing this table later must be performed again. Another advantage of stored results is in reusing them in the context of another path. Namely, it may be the case that no alternative plans for the same goal are available, and an alternative path should be pursued. When switching to an alternative path, the old path results that are still valid can be substituted for appropriate goals.

The strategies for error recovery are dependent on the hardware and software characteristics of available servers, and the environment in which the SIMS system is running. We assume in this work that these strategies have been provided by appropriate domain experts, and the role of SIMS is to provide mechanisms that make use of these strategies to assure efficient error recovery.

5 Formulating Results in the Request Language

There are three problems in formulating the results of a request:

1. Different servers can provide results in different form. We are currently dealing with this problem by transforming all output into a uniform representation. This solution may be

impractical for large data structures, and we are investigating other approaches to this problem.

2. Some of the results are only needed in the process of request satisfaction, and they are not required as a part of the final result. For example, SHIP-ID may only be needed to access the employment schedule database table, and they are not a part of the SITUATION object requested.

3. The components of the request are distinguished from its output form, which can be an arbitrary structure and is a part of the definition of the requested object. For example, already mentioned request for SITUATION consisting of AREA, MAP, SHIPs and EMPSKDs ought to specify that the result is a single SITUATION object with two parts, a MAP and a number of SHIP-DESCRIPTION objects. A SHIP-DESCRIPTION object is itself a complex structure, consisting of the name of the ship (the SHIP attribute) and its employment schedule (EMPSKD attribute).

We deal with the second and third problem by defining *output form* as part of the request. The language of output forms currently specifies the relations between the components of the output object as a tree structure.

6 Summary

Most of the tasks performed by users of information systems involve interaction with multiple software servers. In order to perform such a task with existing methodologies, the user has to formulate it into subtasks that can be performed by servers in the system. For this, he must know what servers are available, and must speak the language of each. Even when formulation is complete, the user has to invoke each server manually, since the output of one server may not quite "fit" the input of another. If a server fails, the user has to find an alternative way to perform the task, and the results obtained before failure are most often lost.

A tool that can accept input request in a server-independent language and can automatically select, sequence and invoke information servers is of great practical value: it relieves the human user of the burden, and it furnishes a novel opportunity for software integration because it permits the "user" to be another computer system. The system we are developing, SIMS, provides just such capabilities.

The main functions of SIMS are: planning which servers to use and how to combine them; monitoring the execution and replanning around errors; and transforming the results into the form requested. The main characteristic of our approach is that we try to balance the trade-offs between efficiency, optimality, and robustness. The planning phase relies on a model of server functionality and utility. The functionality is expressed in a hierarchical relational model, which gives rise to efficient algorithms for finding paths through services. An optimal path is found using the utility model, which characterizes servers in terms of the time/space requirements and the quality of data they provide. Robustness is achieved by specifying multiple paths and multiple servers, and efficiency is achieved by making only minimal modifications to the current path in case of an execution failure.

We have designed an architecture for SIMS, and the initial system implementation in CommonLISP is running on the Symbolics 3600. It is capable of satisfying requests from a Navy domain expressed in a very simple language by combining function servers and simulated database servers. The system computes multiple paths and has mechanisms for switching to another server or to another path. Our short term plans include working to increase the power of the input and output language so that SIMS can interact with real servers. We are also developing criteria for request optimization and mechanisms for request-formulation error recovery and execution error recovery.

References

[ALTE 86] Richard Alterman.
An Adaptive Planner.
Proceedings of the Fifth National Conference on Artificial Intelligence, pages 65-69, (1986).

[BROV 87] Carol A. Broverman and W. Bruce Croft.
Reasoning About Exceptions During Plan Execution Monitoring.
Proceedings of the Sixth National Conference on Artificial Intelligence, pages 190-195, (1987).

[CARL 79] E. Carlson and W. Metz.
A Design for Table-Driven Display Generation and Management Systems.
IBM Technical Report, IBM Research Laboratories, San Hose, California, (1979).

[COKE 87] Harry Coker et al.
The Multi-Model Database System.
Technical Report NPS52-87-026, Naval Postgraduate School, Monterey, California, (1987).

[FEAT 86] Martin S. Feather.
A Survey and Classification of some Program Transformation Approaches and Techniques.
Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, (1986).

[KACZ 86] Thomas Kaczmarek, Lawrence Miller and Norman K. Sondheimer.
INTEGRATED INTERFACES: A Knowledge-Based User Interface Manager ent Approach.
Research Proposal to the Defense Advanced Research Projects Agency, (1986).

[KAEM 86] William F. Kaemmerer and James A. Larson.
A Graph-Oriented Knowledge Representation and Unification Technique for Automatically Selecting and Invoking Software Functions.
Proceedings of the Fifth National Conference on Artificial Intelligence, pages 825-830, (1986).

[LARS 87] Henrik L. Larsen.
Knowledge Representation in IRIS, an Information Retrieval Intermediary System.
Proceedings of the Seventh International Workshop on Expert Systems and their Applications, (1987).

[MAIE 83] David Maier.
The Theory of Relational Databases.
Computer Science Press, Rockwille, Maryland, (1983).

[MARK 81] William Mark.
Representation and Inference in the Consul System.
Proceedings of the Seventh International Joint Conference on Artificial Intelligence, pages 375-381, (1981).

[MOST 85] Jack Mostow and Donald Cohen.
Automating Program Speedup by Deciding what to Cache.
Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pages 165-172, (1985).

[MOTR 80] Amihai Motro and P. Buneman.
Automatically Merging Databases.
Proceedings of the COMPCON Fall 80, the Twentyfirst IEEE International Conference on Distributed computing, (1980).

[MOTR 87] Amihai Motro.
The Design of FLEX: A Tolerant and Cooperative User Interface to Databases.
Proceedings of the Second International Conference on Human-Computer Interaction, (1987).

[NEBE 87] Bernhard Nebel.
Computational Complexity of Terminological Reasoning in BACK.
Technical Report Number 43, Technische Universität Berlin, (1987).

[NECH 85] Robert Neches, William R. Swartout and Johanna Moore.
Explainable (and Maintainable) Expert Systems.
Proceedings of the Ninth International Joint Conference on Artificial Intelligence, pages 382-389, (1985).

[NILS 80] Nils J. Nilsson.
Principles of Artificial Intelligence.
Tioga Publishing Company, Palo Alto, California, (1980).

[SCHM 83] James Schmolze and Thomas Lipkis.
Classification in the KL-ONE Knowledge Representation System.
Proceedings of the Eighth International Joint Conference on Artificial Intelligence, (1983).

[STAL 84] David G. Stallard.
Data Modeling for Natural Language Access.
Proceedings of the First IEEE Conference on Applied Artificial Intelligence, pages 19-24, (1984).

[STEF 86] Mark Stefik and Daniel Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine, Volume VI, Number 4, pages 40-62, (1986).

[STEM 86] David Stemple, Tim Sheard and Ralph Bunker.
Abstract Data Types in Databases: Specification, Manipulation and Access.
Proceedings of the Second IEEE Conference on Data Engineering, pages 590-597, (1986).

[TOU 82] Frederich N. Tou et al.
RABBIT: An Intelligent Database Assistant.
Proceedings of the Second National Conference on Artificial Intelligence, pages 314-318, (1982).

[WILK 84] David E. Wilkins.
Domain-Independent Planning: Representation and Plan Generation.
Artificial Intelligence, Volume 22, pages 269-301, (1984).

[WILK 85] David E. Wilkins.
Recovering from Execution Errors in SIPE.
Technical Report Number 346, Stanford Research Institute, Menlo park, California, (1985).